A Semi-Discrete Matrix Decomposition for Latent Semantic Indexing in Information Retrieval

Tamara G. Kolda
Oak Ridge National Laboratory
and
Dianne P. O'Leary
The University of Maryland

The vast amount of textual information available today is useless unless it can be effectively and efficiently searched. The goal in information retrieval is to find documents that are relevant to a given user query. We can represent a document collection by a matrix whose (i,j) entry is nonzero only if the ith term appears in the jth document; thus each document corresponds to a column vector. The query is also represented as a column vector whose ith term is nonzero only if the ith term appears in the query. We score each document for relevancy by taking its inner product with the query. The highest scoring documents are considered the most relevant. Unfortunately, this method does not necessarily retrieve all relevant documents because it is based on literal term matching. Latent Semantic Indexing (LSI) replaces the document matrix with an approximation generated by the truncated singular value decomposition (SVD). This method has been shown to overcome many difficulties associated with literal term matching. In this paper we propose replacing the SVD with the semi-discrete decomposition (SDD). We will describe the SDD approximation, show how to compute it, and compare the SDD-based LSI method to the SVD-based LSI method. We will show that SDD-based LSI does as well as SVD-based LSI in terms of document retrieval while requiring only one-twentieth the storage and one-half the time to compute each query. We will also show how to update the SDD approximation when documents are added or deleted from the document collection.

Categories and Subject Descriptors: H.3.3 [**Information Systems**]: Information Search and Retrieval; G.1.2 [**Numerical Analysis**]: Approximation

Kolda's work was supported through a fellowship from the the National Physical Sciences Consortium, the National Security Agency, and the University of Maryland. Summer support was provided by the IDA Center for Computing Sciences.
O'Leary's work was supported by the National Science Foundation under Grant CCR-95-03126.

Name: Tamara G. Kolda
Affiliation: Computer Science and Mathematics Division, Oak Ridge National Laboratory
Address: P.O. Box 2008, Bldg. 6012, Oak Ridge, TN 37831-6367 (kolda@msr.epm.ornl.gov)
Name: Dianne P. O'Leary
Affiliation: Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland
Address: College Park, MD 20742 (oleary@cs.umd.edu)

General Terms: Algorithms

Additional Key Words and Phrases: Data mining, latent semantic indexing, semi-discrete decomposition, singular value decomposition, text retrieval

1. INTRODUCTION

The vast amount of textual information available today is useless unless it can be effectively and efficiently searched. The goal in *information retrieval* is to match user information requests, or *queries*, with relevant information items, or *documents*. Examples of information retrieval systems include electronic library catalogs, the `grep` string-matching tool in Unix, and search engines for the World Wide Web such as Alta Vista.

Oftentimes users are searching for documents about a particular concept that may not be accurately described by a list of keywords. For example, a search on a term such as "Mark Twain" is likely to miss some documents about "Samuel Clemens." *We* might know that these are the same people, but most information retrieval systems have no way of knowing. *Latent semantic indexing* (LSI) overcomes this problem by automatically discovering *latent* relationships in the document collection.

Before we discuss LSI further, we need to introduce the *vector space method*: The document collection is represented by an $m \times n$ *term-document matrix* where m is the number of terms and n is the number of documents. Typically this matrix has fewer than 1% nonzero entries. Queries are represented as m-vectors, and a matrix-vector product produces an n-vector of scores that is used to rank the documents in relevance. This method is described in §2.

LSI is based on the vector space method, but the $m \times n$ term-document matrix is replaced with a low-rank approximation generated by the truncated singular value decomposition (SVD). The truncated SVD approximation is the sum of k rank-1 outer products of m-vectors u_i with n-vectors v_i, weighted by scalars σ_i:

$$\sum_{i=1}^{k} \sigma_i u_i v_i^T .$$

Here, k is chosen to be much smaller than m and n. This approximation produces the closest rank-k matrix to the term-document matrix in the Frobenius measure [Golub and Van Loan 1989]. LSI has performed well on both large and small document collections; see, for example, Dumais [1991, 1995]. LSI is described in §3.

Thus far, only the SVD and its relatives, the ULV and URV decompositions [Berry and Fierro 1996], have been used in LSI. We propose using a very different decomposition, originally developed for image compression by O'Leary and Peleg [1983]. In this decomposition, which we call the semi-discrete decomposition (SDD), the matrix is approximated by a sum of rank-1 outer products just as in the SVD, but the m-vectors and n-vectors are restricted to only have entries in the set $\{-1, 0, 1\}$. The decomposition is constructed via a greedy algorithm and is not an optimal decomposition in the sense of producing the best possible approximation in the Frobenius and Euclidean norms; however, the SDD-based LSI method does as well as the SVD-based method in terms of document retrieval while requiring less than one-twentieth the storage and only one-half the query time. The trade-off is that the SDD matrix approximation takes substantially longer to compute; fortunately, this is only a one-time expense. The SDD-based LSI method is described in §4, and computational comparisons with the SVD-based method are given in §5.

In many information retrieval settings, the document collection is frequently updated. Much work has been done on updating the SVD approximation to the term-document matrix [Berry et al. 1995; O'Brien 1994], but it can be as expensive as computing the original SVD. Efficient algorithms for updating the SDD approximation are given in §6.

A preliminary report on the idea was given in Kolda and O'Leary [1997]. Some of the material in this paper is taken from Kolda [1997].

2. THE VECTOR SPACE METHOD

Both the SVD- and the SDD-based LSI methods are extensions of the vector space method, which we describe in this section.

2.1 Creating the Term-Document Matrix

We begin with a collection of textual documents. We determine a list of keywords or *terms* by

(1) creating a list of all words that appear in the documents,

(2) removing words void of semantic content such as "of" and "because" (using the *stop word* list of Frakes [1992]), and

(3) further trimming the list by removing words that appear in only one document.

The remaining words are the terms, which we number from 1 to m. Further discussion of preprocessing techniques can be found in Kolda [1997].

We then create an $m \times n$ *term-document matrix*

$$A = [a_{ij}],$$

where a_{ij} represents the *weight* of term i in document j. See Figure 1 for an example of a 6×4 term-document matrix.

	Document				
Term	1	2	3	4	**Query**
Mark	**15**	0	0	0	**1**
Twain	**15**	0	**20**	0	**1**
Samuel	0	**10**	**5**	0	0
Clemens	0	**20**	**10**	0	0
Purple	0	0	0	**20**	0
Fairy	0	0	0	**15**	0
Score	**30**	0	**20**	0	

Fig. 1. Vector space method

A natural choice of weights is to set $a_{ij} = f_{ij}$, the number of times that term i appears in document j, but more elaborate weighting schemes often yield better performance.

A term weight has three components: local, global, and normalization [Salton and Buckley 1988]. We let

$$a_{ij} = g_i \, t_{ij} \, d_j,$$

where t_{ij} is the local term component (based on information in the jth document only), g_i is the global component (based on information about the use of the ith term throughout the collection), and d_j is the normalization component, specifying whether or not the columns (i.e. the documents) are normalized. Various formulas for each component are given in Tables 1 – 3. In these formulas, χ represents the signum function:

$$\chi(t) = \begin{cases} 1 & \text{if } t > 0, \\ 0 & \text{if } t = 0. \end{cases}$$

Table 1. Local Term Weights

Symbol	Formula for t_{ij}	Description	Reference
b	$\chi(f_{ij})$	Binary	Salton and Buckley [1988]
t	f_{ij}	Term Frequency	Salton and Buckley [1988]
c	$.5\left(\chi(f_{ij}) + \frac{f_{ij}}{\max_k f_{kj}}\right)$	Augmented Normalized Term Frequency	Harman [1992], Salton and Buckley [1988]
l	$\log(f_{ij} + 1)$	Log	Harman [1992]

Table 2. Global Term Weights

Symbol	Formula for g_i	Description	Reference
x	1	No change	Salton and Buckley [1988]
f	$\log\left(\frac{n}{\sum_j \chi(f_{ij})}\right)$	Inverse Document Frequency (IDF)	Salton and Buckley [1988]
p	$\log\left(\frac{n - \sum_j \chi(f_{ij})}{\sum_j \chi(f_{ij})}\right)$	Probabilistic Inverse	Harman [1992], Salton and Buckley [1988]

Table 3. Document Length Normalization

Symbol	Formula for d_j	Description	Reference
x	1	No Change	Salton and Buckley [1988]
n	$\left(\sum_{i=1}^{m} (g_i t_{ij})^2\right)^{-1/2}$	Normal	Salton and Buckley [1988]

The weighting scheme is specified by a three letter string whose letters represent the local, global, and normalization components respectively; for example, using weight `lxn` specifies that

$$a_{ij} = \frac{\log(f_{ij}+1)}{\sqrt{\sum_{k=1}^{m}\left(\log(f_{kj}+1)\right)^2}},$$

that is, log local weights, no global weights, and column normalization.

2.2 Query Creation and Processing

A query is represented as an m-vector

$$q = [q_i],$$

where q_i represents the weight of term i in the query. In order to rank the documents, we compute

$$s = q^T A,$$

and the jth entry of s represents the score of document j. The documents can then be ranked according to their scores, highest to lowest, for relevance to the query. In Figure 1, we show the result of performing a query on "Mark Twain." Notice that document 2 is not recognized as relevant even though it probably should be, since "Mark Twain" was a pseudonym used by Samuel Clemens. This illustrates the problem with literal term matching.

We must also specify a term weighting for the query. This need not be the same as the weighting for the documents [Salton and Buckley 1988]. Here

$$q_i = g_i\,\hat{t}_i,$$

where g_i is computed based on the frequencies of terms in the *document* collection, and $\hat{t}_i$ is computed using the same formulas as for t_{ij} given in Table 1 with f_{ij} replaced by $\hat{f}_i$, the frequency of term i in the query. Normalizing the query vector has no effect on the document rankings, so we never do it. This means the last component of the three-letter query weighting string is always `x`. So, for example, the weighting `cfx` means

$$q_i = \left(.5\,\chi(\hat{f}_i) + .5\left(\frac{\hat{f}_i}{\max_k \hat{f}_k}\right)\right)\log\left(\frac{n}{\sum_{j=1}^{n} f_{ij}}\right).$$

A six-letter string, e.g. `lxn.cfx`, specifies the document and query weights. We will use various weightings in our LSI experiments. The choice of these weightings is a result of more extensive studies presented in Kolda [1997].

3. LSI VIA THE SVD

3.1 Approximating the Term-Document Matrix

In LSI, we use an approximation of the term-document matrix, as generated by the truncated SVD. The SVD decomposes A into a set of $r = \text{rank}(A)$ triplets of left (u_i) and right (v_i) singular vectors and scalar singular values (σ_i):

$$A = \sum_{i=1}^{r} \sigma_i u_i v_i^T.$$

The u_i vectors and v_i vectors each form orthonormal sets, and the positive scalars σ_i are ordered from greatest to least. The SVD is more commonly seen in matrix notation as

$$A = U\Sigma V^T$$

where the columns of U are the left singular vectors, the columns of V are the right singular vectors, and Σ is a diagonal matrix containing the singular values.

The truncated SVD can be used to build a rank-k approximation to A by only using the first $k << r$ triplets; that is,

$$A \approx A_k \equiv \sum_{i=1}^{k} \sigma_i u_i v_i^T .$$

In matrix form, this is written as

$$A \approx A_k \equiv U_k \Sigma_k V_k^T ,$$

where U_k and V_k consist of the first k columns of U and V respectively, and Σ_k is the leading $k \times k$ principal submatrix of Σ. It can be shown that A_k is the best rank-k approximation to A in the Frobenius norm and in the Euclidean norm [Golub and Van Loan 1989]. Each of the n columns of the matrix A_k is a linear combination of the k columns of U_k, and thus we have an implicit clustering of documents based on these principal components

Since the SVD approximation is close to the original matrix, document retrieval based on its use can be expected to be almost as good as document retrieval based on the original matrix. But, in fact, the situation is much better than this. The approximation matrix is a "noisy" version of the original matrix, where the noise was added to reduce the rank (i.e., make the documents appear more similar) while remaining close to the original data. Suppose that "Clemens" and "Twain" often appear together in the document collection. If we then have one document that only mentions "Twain," the approximation may add some noise to the "Clemens" entry as a result of compressing the rank. The amount of noise depends on the size of k. For very small values of k, there is a lot of noise — usually too much — and as k grows, the noise gets smaller until it completely disappears. At some intermediate value of k, we have the optimal amount of noise for recognizing the latent relationship between "Clemens" and "Twain".

Figure 2 shows the result of using a rank-2 truncated SVD to approximate the term-document matrix given in Figure 1.

3.2 Query Processing

We can process queries using our approximation for A:

$$\begin{aligned} s &= q^T A \approx q^T A_k \\ &= q^T U_k \Sigma_k V_k^T \\ &= (q^T U_k \Sigma_k^{\alpha})(\Sigma_k^{1-\alpha} V_k^T) \\ &\equiv \tilde{q}^T \tilde{A}. \end{aligned}$$

The scalar α controls the splitting of the Σ_k matrix and has no effect unless we re-normalize the columns of $\tilde{A}$, that is, after computing the decomposition, we re-scale

	Document				
Term	1	2	3	4	**Query**
Mark	**3.7**	**3.5**	**5.5**	0	**1**
Twain	**11.0**	**10.3**	**16.1**	0	**1**
Samuel	**4.1**	**3.9**	**6.1**	0	0
Clemens	**8.3**	**7.8**	**12.2**	0	0
Purple	0	0	0	**20**	0
Fairy	0	0	0	**15**	0
Score	**14.7**	**13.8**	**21.6**	0	

Fig. 2. LSI via the SVD

each column of $(\Sigma_k^{1-\alpha} V_k^T)$ so that it has Euclidean norm one. We will experiment with various choices for α and re-normalization in § 5.2.

In the example in Figure 2, we do not use re-normalization. Observe that document 2 is now recognized as relevant because "noise" has been added to the "Mark" and "Twain" entries in column 2.

The SVD has been used quite effectively for information retrieval, as documented in numerous reports. We recommend the original LSI paper [Deerwester et al. 1990], a paper reporting the effectiveness of the LSI approach on the TREC-3 dataset [Dumais 1991], and a more mathematical paper by Berry, Dumais, and O'Brien [1995].

4. LSI VIA A SEMI-DISCRETE DECOMPOSITION

4.1 Approximating the Term-Document Matrix

The truncated SVD produces the best rank-k approximation to a matrix, but generally even a very low rank truncated SVD approximation requires more storage than the original matrix if the original matrix is sparse. To save storage (and query time), we propose replacing the truncated SVD by the semi-discrete decomposition (SDD):

$$A_k = \sum_{i=1}^{k} d_i x_i y_i^T,$$

where each m-vector x_i and each n-vector y_i is constrained to have entries from the set $\mathcal{S} = \{-1, 0, 1\}$, and the scalar d_i is any positive number. We can also express this in matrix notation as

$$A_k = X_k D_k Y_k^T,$$

where $X_k = \{x_1 \cdots x_k\}$, $Y_k = \{y_1 \cdots y_k\}$, and $D_k = \text{diag}\{d_1, \cdots, d_k\}$. This decomposition was originally introduced in O'Leary and Peleg [1983]. The SDD does not reproduce A exactly, even if $k = n$, but it uses very little storage with respect to the observed accuracy of the approximation. A rank-k[1] SDD requires the storage of $k(m+n)$ values from the set $\{-1, 0, 1\}$ and k scalars. An element of the set $\{-1, 0, 1\}$ can be expressed using $\log_2 3$ bits, although we use two bits per element for simplicity. The scalars need to be only single precision because the algorithm is

[1] Although the approximation may not be rank-k *algebraically*, it is the sum of k rank-1 matrices.

self-correcting. The SVD, on the other hand, has been computed in double precision accuracy for numerical stability [Paige 1974; Wilkinson 1965] and to keep open the possibility of updating the decomposition if documents are added or deleted from the collection [Berry et al. 1995; O'Brien 1994]. Assuming that double precision scalars require 8 bytes and single precision scalars require 4, and packing 8 bits in a byte, we obtain the following storage comparison between rank-k SVD and SDD approximations to an $m \times n$ matrix:

Method	Component		Total Bytes
SVD	U	km double precision numbers	$8k(m+n+1)$
	V	kn double precision numbers	
	Σ	k double precision numbers	
SDD	X	km numbers from $\{-1,0,1\}$	$4k + \frac{1}{4}k(m+n)$
	Y	kn numbers from $\{-1,0,1\}$	
	D	k single precision numbers	

For equal values of k, the SVD requires nearly 32 times more storage than the SDD. However, we shall see that the SDD rank should be about 50% larger than the SVD rank and results in a 95% reduction in storage.

The SDD approximation is formed iteratively. The remainder of this section comes from O'Leary and Peleg [1983] but is presented here in a slightly different form. Let $A_0 = 0$, and let R_k be the residual matrix at the kth step, that is, $R_k = A - A_{k-1}$. We wish to find a triplet (d_k, x_k, y_k) that solves

$$\min_{\substack{x \in \mathcal{S}^m \\ y \in \mathcal{S}^n \\ d>0}} F_k(d,x,y) \equiv \|R_k - dxy^T\|_F^2. \tag{1}$$

This is a mixed integer programming problem.

We can formulate this as an integer programming problem by eliminating d. For convenience, we temporarily drop the subscript k. We have

$$F(d,x,y) = \sum_{i=1}^{m}\sum_{j=1}^{n}(r_{ij} - dx_i y_j)^2 = \|R\|_F^2 - 2dx^T Ry + d^2\|x\|_2^2\|y\|_2^2.$$

At the optimal solution,

$$\partial F/\partial d = -2x^T Ry + 2d\|x\|_2^2\|y\|_2^2 = 0,$$

so the optimal value, d^*, of d is given by

$$d^* = \frac{x^T Ry}{\|x\|_2^2\|y\|_2^2}.$$

Plugging d^* into F, we get

$$\begin{aligned} F(d^*,x,y) &= \|R\|_F^2 - 2\left(\frac{x^T Ry}{\|x\|_2^2\|y\|_2^2}\right)x^T Ry + \left(\frac{x^T Ry}{\|x\|_2^2\|y\|_2^2}\right)^2\|x\|_2^2\|y\|_2^2 \\ &= \|R\|_F^2 - \frac{(x^T Ry)^2}{\|x\|_2^2\|y\|_2^2}. \end{aligned} \tag{2}$$

Thus (1) is equivalent to

$$\max_{\substack{x\in\mathcal{S}^m\\ y\in\mathcal{S}^n}} \tilde{F}(x,y) \equiv \frac{(x^TRy)^2}{\|x\|_2^2\|y\|_2^2}, \tag{3}$$

which is an integer programming problem with $3^{(m+n)}$ feasible points.

When both m and n are small, we enumerate the feasible points and compute each function value to determine the maximizer. However, as the size of m and/or n grows, the cost of this approach grows exponentially. Rather than trying to solve the problem exactly, we use an *alternating algorithm* to generate an approximate solution. We begin by fixing y and solving (3) for x, we then fix that x and solve (3) for y, we then fix that y and solve (3) for x, and so on.

Solving (3) is very easy when either x or y is fixed. Suppose that y is fixed. Then we must solve

$$\max_{x\in\mathcal{S}^m} \frac{(x^Ts)^2}{\|x\|_2^2}, \tag{4}$$

where $s = Ry/\|y\|_2$ is fixed. Sort the elements of s so that

$$|s_{i_1}| \geq |s_{i_2}| \geq \cdots \geq |s_{i_m}|.$$

If we knew x had exactly J nonzeros, then it is clear that the solution to (4) would be given by

$$x_{i_j} = \begin{cases} \operatorname{sign}(s_{i_j}) & \text{if } 1 \leq j \leq J \\ 0 & \text{if } J+1 \leq j \leq m \end{cases}.$$

Therefore, there are only m possible x-vectors we need to check to determine the optimal solution for (4).

Hence, the O'Leary-Peleg algorithm to find the SDD approximation of rank $k_{\max}$ to an $m \times n$ matrix A is given by

```
(1) Let $R_1 = A$.

(2) Outer Iteration ($k = 1, 2, \ldots, k_{\max}$):

    (a) Choose a starting vector $y$ such that $R_k y \neq 0$.

    (b) Inner Iteration ($i = 1, 2, \ldots, i_{\max}$):
        i.  Fix $y$ and let $x$ solve $\max_{x\in\mathcal{S}^m} \frac{(x^T R_k y)^2}{\|x\|_2^2}$.
        ii. Fix $x$ and let $y$ solve $\max_{y\in\mathcal{S}^n} \frac{(y^T R_k^T x)^2}{\|y\|_2^2}$.
    (c) End Inner Iteration.
    (d) Let $x_k = x$, $y_k = y$, $d_k = \frac{x_k^T R_k y_k}{\|x_k\|_2^2\|y_k\|_2^2}$.

    (e) Let $R_{k+1} = R_k - d_k x_k y_k^T$.

(3) End Outer Iteration.
```

We specify a set number of iterations for the inner loop, but we may use a heuristic stopping criterion instead. From (2) note that

$$\|R_{k+1}\|_F^2 = \|R_k - d_k x_k y_k^T\|_F^2 = \|R_k\|_F^2 - \frac{(x_k^T R_k y_k)^2}{\|x_k\|_2^2 \|y_k\|_2^2}. \tag{5}$$

So for a given (x, y) pair, we can compute exactly what the F-norm of R_{k+1} will be if we accept them. The method proposed in O'Leary and Peleg [1983] to determine when to stop the inner iterations is the following: At the beginning of the inner iterations, set `change = 1`. Then at the end of each inner iteration, compute

$$\begin{aligned} \texttt{newchange} &= \frac{(x^T R_k y)^2}{\|x\|_2^2 \|y\|_2^2}, \text{ and} \\ \texttt{improvement} &= \frac{|\texttt{newchange} - \texttt{change}|}{\texttt{change}}, \\ \texttt{change} &= \texttt{newchange}. \end{aligned}$$

Once `improvement` falls below a given level, say 0.01, we terminate the inner iterations. In other words, we iterate until the improvement in the residual has stagnated. This is the method we use in our tests, and experimentally we found that 0.01 was a good tolerance. As a starting vector for the inner iteration, we use a vector in which every 100th element is one and all the others are zero. It can be shown that, under mild assumptions on the starting guess, we are ensured that $A_k \to A$ as $k \to \infty$; see Kolda [1997] for this and other convergence results. Experiments using a singular vector as a starting guess, or starting guesses with guaranteed convergence, did not produce improvement over this simple minded initialization.

Assuming that we do a fixed number of inner iterations per step, the complexity of the algorithm is $O(k^2(m+n) + m \log m + n \log n)$. In practice, we found that the number of inner iterations to reach the convergence tolerance averaged near 10.

In Figure 3 we show the result of approximating the term document matrix in Figure 1 with a rank-2 SDD approximation. As in the SVD, noise has been added to the "Mark" and "Twain" entries for document 2, revealing the latent relationship.

	Document				
Term	1	2	3	4	**Query**
Mark	**7.9**	**7.9**	**7.9**	0	**1**
Twain	**7.9**	**7.9**	**7.9**	0	**1**
Samuel	**7.9**	**7.9**	**7.9**	0	0
Clemens	**7.9**	**7.9**	**7.9**	0	0
Purple	0	0	0	**17.5**	0
Fairy	0	0	0	**17.5**	0
Score	**15.8**	**15.8**	**15.8**	0	

Fig. 3. LSI via the SDD

4.2 Query Processing

We evaluate queries in much the same way as we did for the SVD, by computing $s = \tilde{q}^T \tilde{A}$, with

$$\tilde{A} = D_k^{1-\alpha} Y_k^T, \quad \tilde{q} = D_k^{\alpha} X_k^T q.$$

Again, we generally re-normalize the columns of $\tilde{A}$.

In Figure 3, the second document is now recognized as relevant, as it was when using the SVD-based LSI.

For decompositions of equal rank, the SDD-based method requires significantly fewer floating-point operations than the SVD-based method to process the query:

Operation	SDD	SVD
Additions	$k(m+n)$	$k(m+n)$
Multiplications	k	$k(1+m+n)$

If we re-normalize the columns of $\tilde{A}$ then each method requires n additional multiplies and storage of n additional floating point numbers.

5. COMPUTATIONAL COMPARISON OF SDD- AND SVD-BASED LSI

In this section, we present computational results comparing the SDD- and SVD-based LSI methods. All tests were run on a Sparc 20. Our code is in C, with the SVD taken from subroutine `las2` in SVDPACKC [Berry et al. 1993].

5.1 Methods of Comparison

We will compare the SDD- and SVD-based LSI methods using three standard test sets; see Table 4. Each test set comes with a collection of documents, a collection of queries, and the "correct answers," that is, a list of relevant documents.

Table 4. Characteristics of the test sets.

	MEDLINE	**CRANFIELD**	**CISI**
Number of Documents:	1033	1399	1460
Number of Queries:	30	225	35
Number of (Indexing) Terms:	5526	4598	5574
Avg No of Terms/Document:	48	57	46
Avg No of Documents/Term:	9	17	12
% Nonzero Entries in Matrix:	0.87	1.24	0.82
Storage for Matrix (MB):	0.4	0.6	0.5
Avg No of Terms/Query:	10	9	7
Avg No Relevant/Query:	23	8	50

We will compare the systems by looking at *average precision*, a standard measure used by the information retrieval community [Harman 1995, Appendix A]. When we evaluate a query, we receive an ordered list of documents. Let r_i denote the number of relevant documents up to and including position i in the ordered list. For each document, we compute two values: recall and precision. The recall at the ith document is the proportion of relevant documents returned so far, that is,

$$\frac{r_i}{r_n}.$$

(Note that r_n is the total number of relevant documents.) The precision at the ith document, p_i, is the proportion of documents returned so far that are relevant, that is,

$$p_i = \frac{r_i}{i}.$$

The *pseudo-precision* at recall level $x \in [0,1]$, $\tilde{p}(x)$, is defined as

$$\tilde{p}(x) = \max\left\{p_i \mid r_i \geq (x \cdot r_n),\ i = 1, \ldots, n\right\}.$$

The N-point *(interpolated) average precision* for a single query is defined as

$$\frac{1}{N}\sum_{i=0}^{N-1} \tilde{p}\left(\frac{i}{N-1}\right).$$

We use 11-point average precision. Since we have multiple queries, we will consider the mean and median average precision over all queries in each data set.

5.2 Parameter Choices

We have two parameter choices to make for the SDD- and SVD-based LSI methods: the choice of the splitting parameter α, and the choice of whether or not to re-normalize the columns of $\tilde{A}$.

Table 5. Mean average precision for the SDD- and SVD-based methods with different parameter choices on the MEDLINE data set with k=100 and weighting `lxn.bpx`.

	SDD		SVD	
	Re-Normalize?		Re-Normalize?	
α	Yes	No	Yes	No
0	62.1	61.2	65.1	64.2
0.5	62.6	61.2	64.7	64.2
-0.5	57.9	61.2	64.7	64.2
1.0	61.7	61.2	64.2	64.2
-1.0	48.6	61.2	62.3	64.2

We investigated these two choices with the SVD- and SDD-based methods on the MEDLINE data set using the weighting `lxn.bpx`. The results are summarized in Table 5. In all further tests, we will use $\alpha = 0.5$ with re-normalization for the SDD-based method and $\alpha = 0$ with re-normalization for the SVD-based method. We experimented using other weightings and other data sets and confirmed that these parameter choices are always best or very close to it.

5.3 Comparisons

We tested the SDD- and SVD-based LSI methods and the vector space method (VS) of Section 2 with a number of weightings. We selected these particular weightings because they performed well for the vector space method [Kolda 1997]. Note that we choose not to use a global weight on the term-document matrix; our studies have shown that global weightings on the term-document matrix have no positive effect on performance. We do use global weightings on the query. This can be thought of as applying the global weighting *after* doing the decomposition. We present mean average precision results in Table 6 using a rank $k = 100$ approximation in each method; this table also includes vector space method results for comparison.

Table 6. Mean average precision results for the SDD-based LSI, SVD-based LSI, and the vector space (VS) methods with k=100.

	MEDLINE			CRANFIELD			CISI		
Weight	SDD	SVD	VS	SDD	SVD	VS	SDD	SVD	VS
lxn.bfx	62.6	64.6	54.6	35.7	**40.4**	45.5	15.6	16.6	17.8
lxn.bpx	**62.6**	**65.1**	**54.6**	35.6	39.9	45.5	15.2	16.9	17.9
lxn.lfx	61.2	64.0	53.7	**35.8**	40.3	45.6	16.0	16.6	18.3
lxn.lpx	61.3	64.3	53.8	35.5	40.1	**45.7**	15.5	16.9	18.4
lxn.tfx	60.9	63.5	53.2	35.7	40.2	45.6	16.3	16.9	**18.4**
lxn.tpx	60.9	63.8	53.4	35.4	39.9	45.6	15.7	17.0	18.4
cxx.bpx	57.9	59.6	51.9	32.9	38.9	43.4	17.1	**17.9**	17.6
cxn.bfx	58.4	62.5	53.6	33.1	38.7	44.1	17.8	16.5	17.5
cxn.bpx	58.4	63.0	53.6	32.6	38.7	43.4	**18.1**	17.6	17.5
cxn.tfx	56.8	61.5	52.5	33.3	38.8	43.9	17.1	16.9	18.3
cxn.tpx	57.0	61.8	52.5	32.7	38.2	43.3	17.1	17.7	18.3

To continue our comparisons, we select a "best" weighting for each data set. In Table 6 we have highlighted the "best" results for each data set in boldface type. We will use only these weightings for the remainder of the paper.

In Figure 4, we present results for the MEDLINE data. The upper right graph plots the mean average precision vs. query time for approximations of increasing rank. For example, the left-most asterisk corresponds to the rank-10 SDD, the next asterisk corresponds to the rank-20 SDD, and so on. The SVD results are presented in the same manner using circles. The vertical dotted line shows the query time for the vector space method, and the horizontal dotted line shows the mean average precision for the vector space method. Note that each LSI method reaches a mean average precision peak and then declines, asymptotically approaching the performance of the vector space method; the peak corresponds to when we have added just the right amount of "noise." The SDD-based method peaks at a mean average precision of 63.6, corresponding to a query time of 3.4 seconds using a rank-140 approximation. To be that fast in terms of query time, the SVD-based method can only use a rank-20 approximation that achieves a mean average precision of 51.8. At its peak mean average precision of 65.5, the SVD-based method has a query time of 6.3 seconds using a rank-110 approximation. Note that both methods require more time for the query than the vector space method, but this is the trade-off for increased performance in terms of average precision. The upper-right graph is the same, only this time we are using median average precision rather than mean average precision.

The middle left graph in Figure 4 plots mean average precision against storage for the decomposition. Again, the left-most asterisk corresponds to the rank-10 SDD, the next asterisk to the right corresponds to the rank-20 SDD and so on. The SVD method is plotted in the same manner using circles. Observe that there are only three data points for the SVD; this is because the remaining points are past the

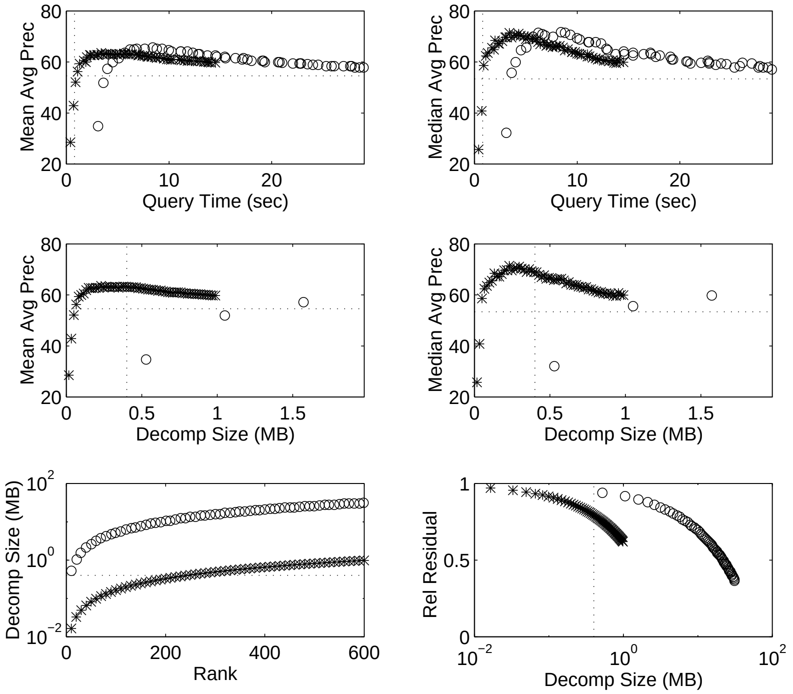

Fig. 4. A comparison of the SDD- (∗) and SVD-based (∘) LSI method on the MEDLINE data set. We plot 60 data points for each graph, corresponding to $k = 10, 20, \ldots, 600$. The dotted lines show the corresponding data for the vector space method.

edge of the graph[2]. The horizontal dotted line shows the mean average precision for the vector space method, and the vertical dotted line shows how much space is required to store the original term-document matrix. Observe that at its mean average precision peak, the SDD-based method needs only 0.2 megabytes (MB) of storage, only half that required by the original matrix (0.4 MB). Conversely, the SVD-based method requires over 5 MB of storage at its mean average precision peak. The middle right graph is the same as the graph on the left except that we are using median average precision rather than mean average precision.

The bottom left graph in Figure 4 plots decomposition size against the rank of the matrix. Note that the y-axis is logarithmic. The asterisks and circles are defined as before, and the vertical dotted line indicates how much storage the original matrix required.

The bottom right graph in Figure 4 plots the norm of the relative residual ($\|R_k\|_F / \|R_0\|_F$) against storage. Note that the x-axis is logarithmic. The asterisks

[2] Storage space can be economized for the SVD by using lower precision in U_k and V_k. For instance, if 8-bit fixed-point precision is used, the mean average precision is 64.5 rather than the 65.5 of double precision. Beyond this point, however, the mean average precision drops rapidly: 60.2, 32.3, and 6.5 for 6, 4, and 2-bit precision, respectively. Even if low precision is used in retrieval, a higher precision version would need to be preserved if future updates to the document collection are expected.

and circles are defined as before, and the horizontal dotted line indicates how much room the original matrix required. The primary observation we make from this graph is that if we are given SDD and SVD approximations of equal storage size, the SDD approximation will have a lower residual.

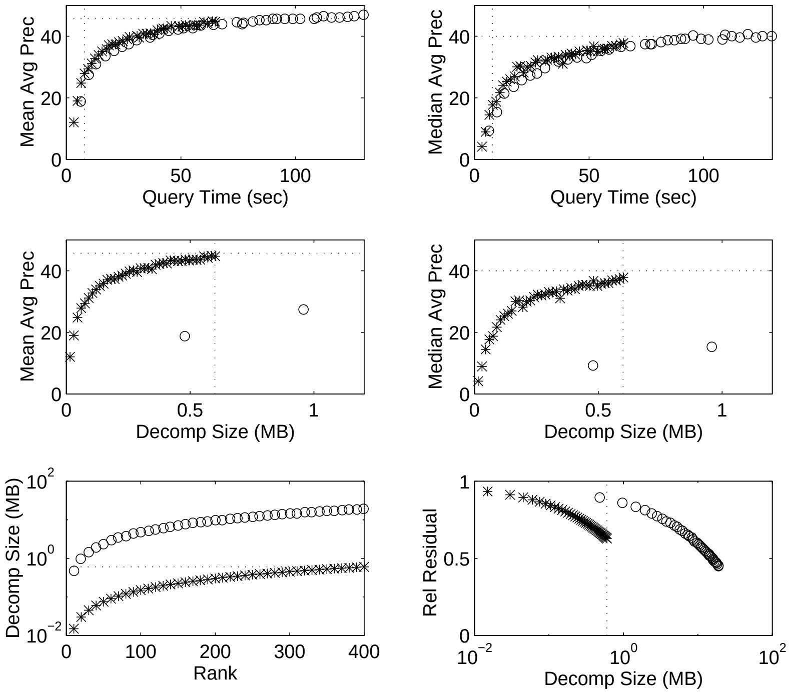

Fig. 5. A comparison of the SDD- ($*$) and SVD-based ($\circ$) LSI methods on the CRANFIELD data set. We plot 40 data points for each graph, corresponding to $k = 10, 20, \ldots, 400$. The dotted lines show the corresponding data for the vector space method.

Figure 5 shows the same series of graphs for the CRANFIELD dataset. This dataset is troublesome for LSI techniques; they do not do as well as the vector space method. From the upper two graphs in Figure 5 we see that, for equal query times, the SDD method does as well as the SVD-based method in terms of average precision. Storage-wise, we again see the SDD-based method is much more economical than the SVD-based method. The bottom two graphs look nearly identical to those in Figure 4.

Lastly, Figure 6 shows the same series of graphs on the CISI dataset. Here the SDD-based method peaks slightly higher than the SVD-based method as shown in the two upper graphs. The SDD-based method peaks at a mean average precision of 19.1 at 4.3 seconds using a rank-140 SDD. If we restrict the SVD-based method to only 4.3 seconds of query time, it can only use a rank-30 SVD which achieves a mean average precision of 15.2. At its peak, the SVD-based method reached a mean average precision of 18.3 using a rank-90 SVD. In terms of storage, at its peak the SDD uses less than half the storage (0.2 MB) of the original matrix (0.5

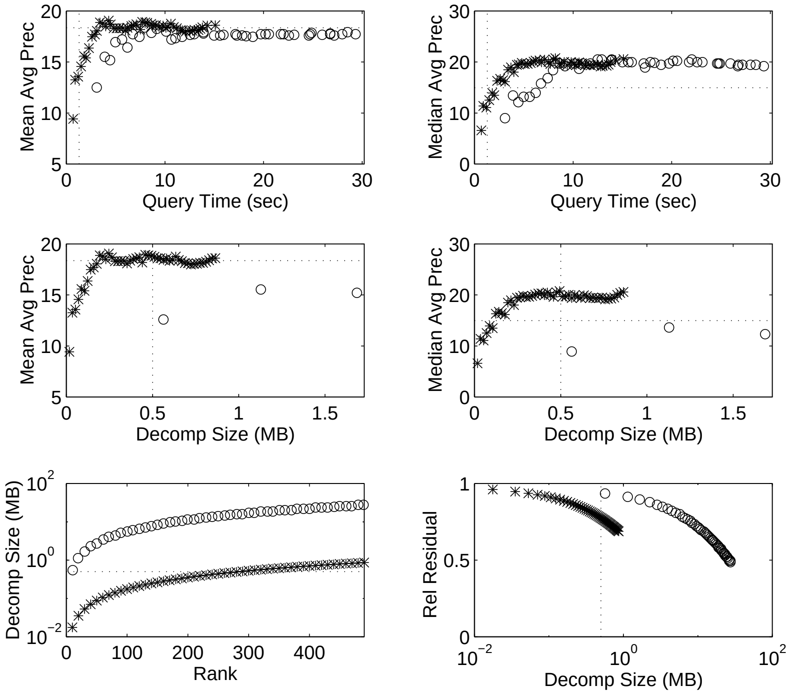

Fig. 6. A comparison of the SDD- ($*$) and SVD-based ($\circ$) LSI methods on the CISI data set. We plot 49 data points for each graph, corresponding to $k = 10, 20, \ldots, 490$. The dotted lines show the corresponding data for the vector space method.

MB). This means that we get compression as well as an increase in performance. The bottom two graphs again look nearly identical to those in Figure 4.

Table 7. Comparison of the SDD- and SVD-based methods at their respective peaks.

	MEDLINE		CRANFIELD		CISI	
	SDD	SVD	SDD	SVD	SDD	SVD
Avg Qry Time (Sec)	**0.11**	0.21	**0.28**	0.65	**0.12**	0.18
Dimension (k)	140	110	390	400	140	90
Mean Avg Prec	63.6	**65.5**	44.9	**47.0**	**19.1**	18.3
Median Avg Prec	70.4	**71.0**	37.3	**40.9**	**19.4**	18.5
Avg in Top Ten	7.17	**7.43**	2.65	**2.67**	**2.51**	2.49
Storage (MB)	**0.2**	5.8	**0.6**	19.2	**0.2**	5.1
Decomp Time (Sec)	245	**54**	1314	**641**	279	**54**
Rel Resid Norm	0.85	0.78	0.63	0.45	0.85	0.81

Table 7 compares the two methods at their respective mean average precision peaks. The first row lists the average time to complete a single query. The SDD-based method is approximately twice as fast as the SVD-based method. The second row lists the rank of the approximation. Although the SDD method saves much less data per vector, it requires only about 50% more vectors than the SVD. The third and fourth rows list the mean and median average precisions respectively. We claim that the methods are basically equal in terms of these measures. To give a more concrete idea of what a user might expect, the fifth row lists the average number of relevant documents returned in the top ten. Again the two methods are nearly equal with respect to this measure. The sixth row compares the storage requirements of the two methods. The SVD approximation requires over twenty times more storage than the SDD approximation. The seventh row shows that it takes about five times longer to compute the SDD than the SVD; however, this is only a one time expense. The last row gives the norm of the relative residual. Note that for both MEDLINE and CISI, this value is around 0.80. In the CRANFIELD data, neither method really peaks. The relative residual may give us a clue in determining how good our approximation should be, that is, what value of k we should choose. Since we know the norm of the residual as a by-product of the approximation, we can easily track the stopping criterion.

To summarize, SDD-based LSI retrieves documents as well as SVD-based LSI, requires only about half the query time, and requires less than one-twentieth the storage. The only disadvantage of the SDD-based method is that computing the SDD approximation takes five times as long as computing the SVD approximation. The SVD is rather difficult to update when the document collection changes, but in the next section we discuss how easy it is to update the SDD, and, in the process, develop a more economical way to compute the initial SDD.

6. MODIFYING THE SDD WHEN THE DOCUMENT COLLECTION CHANGES

Thus far we have discussed the usefulness of the SDD on a fixed document collection. In practice, it is common for the document collection to be dynamic: new documents are added, and old documents are removed. In this section, we will focus on the problem of modifying a SDD decomposition when the document collection changes.

SVD-updating has been studied by O'Brien [1994] and [Berry et al. 1995]. The authors report that updating the SVD takes almost as much time as re-computing it, but that it requires less memory. The authors' methods are similar to what we do in Method 1 in the next section.

6.1 Adding or Deleting Documents

Suppose that we have an SDD approximation for a document collection and then wish to add more documents. Rather than compute a new approximation, we will use the approximation from the original document collection to generate a new approximation for the enlarged collection.

Let m_1 and n_1 be the number of terms and documents in the original collection, n_2 be the number of documents added, and m_2 be the number of *new* terms[3]. Let the new document collection be represented as

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

where

[3] Recall that a term is any word that appears at least twice in the collection and is not a stop word. The addition of new documents may add new terms, some of which may have appeared once in the original document collection.

—A_{11} is an $m_1 \times n_1$ matrix representing the original document collection,

—A_{12} in an $m_1 \times n_2$ matrix representing the new documents indexed by the m_1 terms used in the original collection,

—A_{21} is an $m_2 \times n_1$ matrix representing the original documents indexed by the newly introduced terms, and

—A_{22} is an $m_2 \times n_2$ matrix representing the new documents indexed by the newly introduced terms.

Recall that we are not using global weighting on the term-document matrix, so A_{11} will not change. We will discuss what to do if there are global weights in the next subsection.

Assume that $X^{(1)}$, $D^{(1)}$, and $Y^{(1)}$ are the components of the SDD approximation for A_{11}. We propose two methods for updating this decomposition. Each methods is a two-step process: In the first step, we incorporate the new documents using the existing terms, and in the second step, we incorporate the new terms (for both old and new documents).

Method 1: Append rows to $Y^{(1)}$ and $X^{(1)}$. The simplest update method is to keep the existing decomposition fixed and just append new rows corresponding to the new terms and documents. The D will not be recomputed, so the final D is given by

$$D = D^{(1)}.$$

To incorporate the documents (the first step), we want to find $Y^{(2)} \in \mathcal{S}^{n_2 \times k}$ such that

$$\begin{bmatrix} A_{11} & A_{12} \end{bmatrix} \approx X^{(1)} D \begin{bmatrix} Y^{(1)} \\ Y^{(2)} \end{bmatrix}^T .$$

Let $k_{\max}$ be the rank of the decomposition desired; generally this is the same as the rank of the original decomposition. For each value of $k = 1, \ldots, k_{\max}$, we must find the vector y that solves

$$\min_{y \in \mathcal{S}^{n_2}} \|A^{(c)} - dxy^T\|_F,$$

where $A^{(c)} = A_{12} - X^{(1)}_{k-1} D_{k-1} (Y^{(2)}_{k-1})^T$, x is the kth column of $X^{(1)}$, and d is the kth diagonal element of D. We never access A_{11}, and this may be useful in some situations. The solution y becomes the kth column of $Y^{(2)}$. The final Y is given by

$$Y = \begin{bmatrix} Y^{(1)} \\ Y^{(2)} \end{bmatrix} .$$

To incorporate the terms, we want to find $X^{(2)} \in \mathcal{S}^{m_2 \times k}$ such that

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \approx \begin{bmatrix} X^{(1)} \\ X^{(2)} \end{bmatrix} DY^T.$$

We find $X^{(2)}$ in a analogous way to finding $Y^{(2)}$. For each $k = 1, \ldots, k_{\max}$, we must find the vector x that solves

$$\min_{x \in \mathcal{S}^{m_2}} \|A^{(c)} - dxy^T\|_F,$$

where $A^{(c)} = \left[\, A_{21} \;\; A_{22} \,\right] - X^{(2)}_{k-1} D_{k-1} (Y_{k-1})^T$, y is the kth column of Y, and d is the kth diagonal element of D. Again, we never access A_{11} for this computation. The final X is given by

$$X = \begin{bmatrix} X^{(1)} \\ X^{(2)} \end{bmatrix}.$$

Method 2: Re-Compute Y and D, then X and D. Another possible method is to completely re-compute Y and D (holding X fixed) to incorporate the documents, and then recompute X and D, holding Y fixed.

Specifically, to incorporate the documents, we first want to find $D^{(2)}$ and Y such that

$$\left[\, A_{11} \;\; A_{12} \,\right] \approx X^{(1)} D^{(2)} Y,$$

where Y has no superscript because it will be the final Y.

To do this, let $k_{\max}$ be the rank of the decomposition desired. For each $k = 1, \ldots, k_{\max}$, we must find the d and y that solve

$$\min_{\substack{d>0 \\ y \in \mathcal{S}^n}} \|A^{(c)} - dxy^T\|_F,$$

where $A^{(c)} = A - X^{(1)}_{k-1} D^{(2)}_{k-1} Y^T_{k-1}$ and x is the kth column of $X^{(1)}$. The solutions d and y become the kth diagonal element of $D^{(2)}$ and the kth column of Y respectively.

To incorporate the documents, we wish to find X and D such that

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \approx XDY^T.$$

This is similar to how we computed Y and $D^{(2)}$ in the first step. For each $k = 1, \ldots, k_{\max}$, we must find the d and x that solve

$$\min_{\substack{d>0 \\ x \in \mathcal{S}^m}} \|A^{(c)} - dxy^T\|_F,$$

where $A^{(c)} = A - X_{k-1} D_{k-1} Y^T_{k-1}$ and y is the kth column of Y. The solutions d and x become the kth diagonal element of D and the kth column of X respectively.

Neither method has any inner iterations, and so both are fast. We tried each update method on a collection of tests derived from the MEDLINE data. We split the MEDLINE document collection into two groups. We did a decomposition on the first group of documents with $k = 100$, then added the second group of documents to the collection, and updated the decomposition via each of the two update methods. The results are summarized in Table 8. The second method has better average precision, as should be expected since we are allowing more to change. For the second method, the decrease in mean average precision is not very great when we add only a small number of documents. As the proportion of new documents to old documents grows, however, performance worsens to the point that it is worse than the vector space method. Note, however, that great savings in computation time can be achieved by adding documents incrementally rather than performing the SDD on a large document collection.

Table 8. Comparison of two update methods on the MEDLINE data set with $k = 100$.

Documents		Decomp	Method 1		Method 2	
			Time	Mean	Time	Mean
Old	New	Time (Sec)	(Sec)	Avg Prec	(Sec)	Avg Prec
1033	–	150.5	–	62.18	–	62.18
929	104	138.3	10.5	60.10	13.8	61.83
826	207	122.1	10.4	58.44	13.7	61.80
723	310	103.6	10.2	54.59	13.4	62.46
619	414	94.2	10.2	47.70	13.2	59.28
516	517	77.5	10.1	39.11	12.9	58.76
413	620	60.7	9.9	34.00	12.6	58.83
309	724	45.6	9.5	18.98	12.1	57.19
206	827	26.2	9.6	18.50	11.7	52.29
103	930	14.9	9.4	16.26	11.1	51.38

If we wish to delete terms or documents, we simply delete the corresponding rows in the X and Y matrices. For the SVD, this operation is much more complicated, since orthogonality must be restored.

6.2 Iterative Improvement of the Decomposition

If we have an existing decomposition, perhaps resulting from adding and/or deleting documents and terms, we may wish to improve on this decomposition without recomputing it. We consider two approaches:

Improvement 1: Partial Re-Computation. In order to improve on the decomposition, we could reduce its rank by deleting, say, 10% of the vectors and then recompute them using our original algorithm. This method's main disadvantage is that it can be expensive in time. If performed on the original decomposition, it has no effect.

Improvement 2: Fix and Compute. This method is derived from the second update method. We fix the current X and re-compute the Y and D; we then fix the current Y and re-compute the X and D. This method is very fast because there are no inner iterations. This can be repeated to further improve the results. If applied to an original decomposition, it would change it.

We took the decompositions resulting from the second update method in the last subsection and applied the improvement methods to them. We have a rank-100 decomposition. For the first improvement method, we re-computed 10 dimensions. For the second improvement method, we applied the method once. The results are summarized in Table 9. If we have added only a few documents, neither method is very helpful. On the other hand, if we have added many documents, then the second method is much better. The first method could be improved by re-computing more dimensions, but this would quickly become too expensive. The second method greatly improves poor decompositions and is relatively inexpensive. It can be applied repeatedly to further improve the decomposition.

If we choose to use global weighting on the term-document matrix and the global weights change as a result of adding new documents, we suggest the following procedure. First use improvement steps to improve the SDD decomposition of

Table 9. Comparison of two improvement methods on the MEDLINE data set with $k = 100$.

			Improvement 1		Improvement 2	
Documents		Prev Mean	Time	Mean	Time	Mean
Old	New	Avg Prec	(Sec)	Avg Prec	(Sec)	Avg Prec
1033	–	62.16	22.9	62.16	13.5	61.85
929	104	61.83	20.8	61.45	13.4	61.22
826	207	61.80	19.8	61.51	13.5	62.03
723	310	62.46	21.6	61.91	13.4	61.89
619	414	59.28	19.6	58.70	13.6	61.42
516	517	58.76	19.2	59.43	13.5	59.32
413	620	58.83	20.2	59.68	13.4	61.55
309	724	57.19	20.1	57.94	13.6	59.59
206	827	52.29	21.2	54.35	13.4	57.63
103	930	51.38	22.7	53.88	13.4	56.46

A_{11} with the new global weights, and then proceed with the update procedure as normal.

7. CONCLUSIONS

We have introduced a semi-discrete matrix decomposition for use in LSI. For equal query times, the SDD-based LSI method performs as well as the original SVD-based LSI method. The advantages of the SDD-based method are that the decomposition takes very little storage and the queries are faster; the disadvantage is that the time to form the decomposition is large. Since decomposition is a one-time expense, we believe that the SDD-based LSI method will be quite useful in application.

We have also introduced methods to dynamically update the SDD decomposition if the document collection changes, as well as methods to improve the decomposition if it is found to be inadequate. Updating the SDD is much easier than updating the SVD. Using these updating techniques, the initial decomposition time for the SDD can be greatly reduced without much reduction in average precision.

Open questions remain: in particular, how the algorithm would behave on more extensive document collections, whether a large number of incremental updates to the document collection can be tolerated, and how the algorithm compares to other approaches, such as the INQUERY retrieval system [Broglio et al. 1995], random sampling matrix multiplication algorithms [Cohen and Lewis 1997], and conventional inverted file based retrieval [Witten et al. 1994].

ACKNOWLEDGMENTS

We are grateful to Duncan Buell, John Conroy, Kenneth Kolda, Steven Kratzer, Joseph McCloskey, Douglas Oard, David D. Lewis, and the referees for helpful comments.

REFERENCES

Berry, M., Do, T., O'Brien, G., Krishna, V., and Varadhan, S. 1993. SVDPACKC (version 1.0) users' guide. Technical Report CS-93-194, Department of Computer Science, University of Tennessee, Knoxville, TN 37996.

Berry, M. W., Dumais, S. T., and O'Brien, G. W. 1995. Using linear algebra for intelligent information retrieval. *SIAM Review 37*, 573–595.

Berry, M. W. and Fierro, R. D. 1996. Low-rank orthogonal decompositions for information retrieval applications. *Numerical Linear Algebra with Applications 1*, 1–27.

Broglio, J., Callan, J. P., Croft, W. B., and Nachbar, D. N. 1995. Document retrieval and routing using the INQUERY system. In D. K. Harman Ed., *The Third Text REtrieval Conference* (TREC-3) (1995). NIST Special Publication 500-225. `http://potomac.ncsl.nist.gov/TREC`.

Cohen, E. and Lewis, D. D. 1997. Approximating matrix multiplication for pattern recognition tasks. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms* (1997), pp. 682–691. SIAM, Philadelphia, PA.

Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R. 1990. Indexing by latent semantic analysis. *Journal of the Society for Information Science 41*, 391–407.

Dumais, S. 1991. Improving the retrieval of information from external sources. *Behavior Research Methods, Instruments, & Computers 23*, 229–236.

Dumais, S. T. 1995. Latent semantic indexing (LSI): TREC-3 report. In D. K. Harman Ed., *The Third Text REtrieval Conference* (TREC-3) (1995), pp. 219–230. NIST Special Publication 500-225. `http://potomac.ncsl.nist.gov/TREC`.

Frakes, W. B. 1992. Descent property and global convergence of the Fletcher-Reeves method with inexact line search. In W. B. Frakes and R. Baeza-Yates Eds., *Information Retrieval: Data Structures & Algorithms*, pp. 131–151. Prentice Hall, Englewood Cliffs, NJ 07632.

Golub, G. H. and Van Loan, C. F. 1989. *Matrix Computations* (2nd ed.). The Johns Hopkins University Press, Baltimore.

Harman, D. 1992. Ranking algorithms. In W. B. Frakes and R. Baeza-Yates Eds., *Information Retrieval: Data Structures & Algorithms*, pp. 363–392. Prentice Hall, Englewood Cliffs, NJ 07632.

Harman, D. K. Ed. 1995. *The Third Text REtrieval Conference* (TREC-3) (1995). NIST Special Publication 500-225. `http://potomac.ncsl.nist.gov/TREC`.

Kolda, T. G. 1997. *Limited-Memory Matrix Methods with Applications*. Ph. D. thesis, Applied Mathematics Program, University of Maryland, College Park, MD 20742. Also available as Department of Computer Science Technical Report CS-TR-3806. `http://www.cs.umd.edu/TR/UMCP-CSD:CS-TR-3806`.

Kolda, T. G. and O'Leary, D. P. 1997. Latent semantic indexing via a semidiscrete matrix decomposition. In G. Cybenko, D. P. O'Leary, and J. Rissanen Eds., *The Mathematics of Information Coding, Extraction and Distribution*, IMA Volumes in Math. and Its Applics. Springer-Verlag. To appear. Also available as University of Maryland Department of Computer Science Technical Report CS-TR-3717. `http://www.cs.umd.edu/TR/UMCP-CSD:CS-TR-3717`.

O'Brien, G. 1994. Information management tools for updating an SVD-encoded indexing scheme. Master's thesis, Department of Computer Science, University of Tennessee, Knoxville, TN 37996. Also available as Department of Computer Science Technical Report UT-CS-94-258.

O'Leary, D. P. and Peleg, S. 1983. Digital image compression by outer product expansion. *IEEE Trans. on Communications 31*, 441–444.

Paige, C. C. 1974. Bidiagonalization of matrices and solution of linear equations. *SIAM J. Numer. Anal. 11*, 197–209.

Salton, G. and Buckley, C. 1988. Term weighting approaches in automatic text retrieval. *Information Processing and Management 24*, 513–523.

Wilkinson, J. H. 1965. *The Algebraic Eigenvalue Problem*. Clarendon Press, Oxford.

Witten, I. H., Moffat, A., and Bell, T. C. 1994. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Von Nostrand Reinhold, New York.